CHAPTER 6

Using MongoDB Shell

> *"mongo shell comes with the standard distribution of MongoDB. It offers a JavaScript environment with complete access to the language and the standard functions. It provides a full interface for the MongoDB database."*

In this chapter, you learn the basics of the mongo shell and how to use it to manage MongoDB documents. Before you delve into creating applications that interact with the database, it is important to understand how the MongoDB shell works.

There's no better way to get a feel for a MongoDB database than to get started with the MongoDB shell. The MongoDB shell introduction has been divided into three parts in order to make it easier for the readers to grasp and practice the concepts.

The first section covers the basic features of the database, including the basic CRUD operators. The next section covers advanced querying. The last section of the chapter explains the two ways of storing and retrieving data: embedding and referencing.

Basic Querying

This section will briefly discuss the CRUD operations (Create, Read, Update, and Delete). Using basic examples and exercises, you will learn how these operations are performed in MongoDB. Also, you will understand how queries are executed in MongoDB.

In contrast to traditional SQL, which is used for querying, MongoDB uses its own JSON-like query language to retrieve information from the stored data.

After the successful installation of MongoDB, as explained in Chapter 5, you will navigate to the directory `C:\practicalmongodb\bin\`. This folder has all of the executables for running MongoDB.

The MongoDB shell can be started by executing the mongo executable.

The first step is always to start the database server. Open the command prompt (by running it as administrator) and issue the command `CD \`.

Next, run the command `C:\practicalmongodb\bin\mongod.exe`. (If the installation is in some other folder, the path will change accordingly. For the examples in this chapter, the installation is in the `C:\practicalmongodb` folder.) This will start the database server.

```
C:\>c:\practicalmongodb\bin\mongod.exe
2015-07-06T02:29:24.501-0700 I CONTROL  Hotfix KB2731284 or later update is insalled, no
need to zero-out data files
2015-07-06T02:29:24.522-0700 I JOURNAL  [initandlisten] journal dir=c:\data\db\ournal
.....................................................
2015-07-06T02:29:24.575-0700 I CONTROL  [initandlisten] MongoDB starting : pid=384
port=27017 dbpath=c:\data\db\ 64-bit host=ANC09
```


S.G. Edward and N. Sabharwal, *Practical MongoDB*, DOI 10.1007/978-1-4842-0647-8_6

```
2015-07-06T02:29:24.575-0700 I CONTROL  [initandlisten] targetMinOS: Windows 7/windows
Server 2008 R2
2015-07-06T02:29:24.575-0700 I CONTROL  [initandlisten] db version v3.0.4
2015-07-06T02:29:24.575-0700 I CONTROL  [initandlisten] OpenSSL version: OpenSSL1.0.1j-fips
19 Mar 2015
2015-07-06T02:29:24.575-0700 I CONTROL  [initandlisten] build info: windows sys
getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Service Pack 1')
BOOST_LIB_VERSION=1_49
2015-07-06T02:29:24.575-0700 I CONTROL  [initandlisten] allocator: system
2015-07-06T02:29:24.575-0700 I CONTROL  [initandlisten] options: {}
2015-07-06T02:29:24.584-0700 I NETWORK  [initandlisten] waiting for connections on port 27017
```

MongoDB by default listens for any incoming connections on port 27017 of the localhost interface.

Now that the database server is started, you can start issuing commands to the server using the mongo shell.

Before you look at the mongo shell, let's briefly look at how to use the import/export tool to import and export data in and out of the MongoDB database.

First, create a CSV file to hold the records of students with the following structure:

```
Name, Gender, Class, Score, Age.
```

Sample data of the CSV is shown in Figure 6-1.

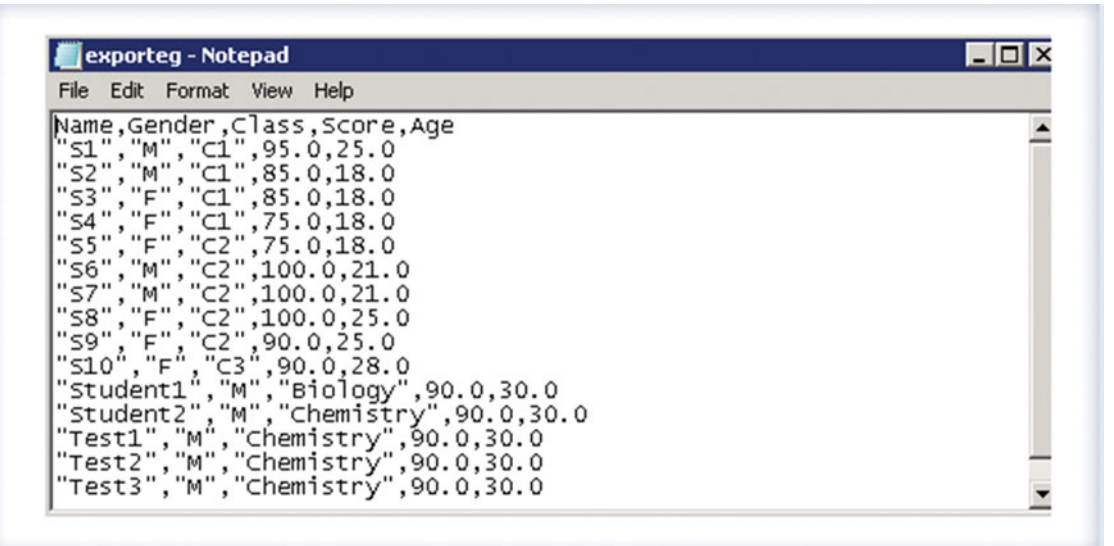

Figure 6-1. *Sample CSV file*

Next, import the data from the MongoDB database to a new collection in order to look at how the import tool works.

Open the command prompt by *running it as an administrator*. The following command is used to get help on the `import` command:

```
C:\>c:\practicalmongodb\bin\mongoimport.exe --help
Import CSV, TSV or JSON data into MongoDB.
When importing JSON documents, each document must be a separate line of the input file.
Example:
  mongoimport --host myhost --db my_cms --collection docs < mydocfile.json
....
C:\>
```

Issue the following command to import the data from the file `exporteg.csv` to a new collection called `importeg` in the `MyDB` database:

```
C:\>c:\practicalmongodb\bin\mongoimport.exe --host localhost --db mydb --collection
importeg --type csv --file c:\exporteg.csv --headerline
2015-07-06T01:53:23.537-0700    connected to: localhost
2015-07-06T01:53:23.608-0700    imported 15 documents
```

In order to validate whether the collection is created and the data is imported, you connect to the database (which is `localhost` in this case) using mongo shell, and you issue commands to validate whether the collection exists or not.

To start the mongo shell, run command prompt as administrator and issue the command `C:\PracticalMongoDB\bin\mongo.exe` (the path will vary based on the installation folder; in this example, the folder is `C:\PracticalMongoDB\`), and press Enter.

This by default connects to the `localhost` database server which is listening on port 27017.

```
C:\>c:\practicalmongodb\bin\mongo.exe
MongoDB shell version: 3.0.4
connecting to: test
> use mydb
switched to db mydb
> show collections
importeg
system.indexes
> db.importeg.find()
{ "_id" : ObjectId("5450af58c770b7161eefd31d"), "Name" : "S1", "Gender" : "M",
"Class" : "C1", "Score" : 95, "Age" : 25 }
.......

{ "_id" : ObjectId("5450af59c770b7161eefd31e"), "Name" : "S2", "Gender" : "M",
"Class" : "C1", "Score" : 85, "Age" : 18 }
>
```

In brief, what you are doing here is

1. Connecting to the mongo shell
2. Switching to your database, which is `MyDB` in this case
3. Checking for the collections that exist in the `MyDB` database using `show collections`.

4. Checking the count of the collection that you imported using the import tool.
5. Finally, executing the `find()` command to check for the data in the new collection.

To connect to different hosts and ports, `-host` *and* `-port` *can be used along with the command.*

As you can see in Figure 6-1, by default the database `test` is used for context.

At any point in time, executing the `db` command will show the current database to which the shell is connected:

```
> db
test
>
```

To display all the database names, you can run the `show dbs` command. Executing this command will list all of the databases for the connected server.

```
> show dbs
```

At any point, help can be accesses using the `help()` command.

```
> help
        db.help()                    help on db methods
        db.mycoll.help()             help on collection methods
        sh.help()                    sharding helpers
        rs.help()                    replica set helpers
        help admin                   administrative help
        help connect                 connecting to a db help
        help keys                    key shortcuts
        help misc                    misc things to know
        help mr                      mapreduce
        show dbs                     show database names
        show collections             show collections in current database
        show users                   show users in current database
.............
        exit                         quit the mongo shell
```

As shown above, if you need help on any of the methods of `db` or `collection`, you can use `db.help()` or `db.<CollectionName>.help()`. For example, if you need help on the `db` command, execute `db.help()`.

```
> db.help()
DB methods:
        db.addUser(userDocument)
...
        db.shutdownServer()
        db.stats()
        db.version() current version of the server
>
```

Until now you have been using the default `test` db. The command `use <newdbname>` can be used to switch to a new database.

```
> use mydb
switched to db mydb
```

Before you start your exploration, let's first briefly look at MongoDB terminology and concepts corresponding to SQL terminology and concepts. This is summarized in Table 6-1.

Table 6-1. *SQL and MongoDB Terminology*

SQL	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Joins within table	Embedding and referencing
Primary Key: A column or set of columns can be specified	Primary Key: Automatically set to _id field

Let's start your exploration of the options for querying in MongoDB. Switch to the `MYDBPOC` database.

```
> use mydbpoc
switched to db mydbpoc
>
```

This switches the context from `test` to `MYDBPOC`. The same can be confirmed using the `db` command.

```
> db
mydbpoc
>
```

Although the context is switched to `MYDBPOC`, the database name will not appear if the `show dbs` command is issued because MongoDB doesn't create a database until data is inserted into the database. This is in keeping with MongoDB's dynamic approach to data facilitating, dynamic namespace allocation, and a simplified and accelerated development process. If you issue the `show dbs` command at this point, it will not list the `MYDBPOC` database in the list of databases because the database is not created until data is inserted into the database.

The following example assumes a polymorphic collection named `users` which contains documents of the following two prototypes:

```
{
_id: ObjectID(),
FName: "First Name",
LName: "Last Name",
Age: 30, Gender: "M",
Country: "Country"
}
and
```

```
{
_id: ObjectID(),
Name: "Full Name",
Age: 30,
Gender: "M",
Country: "Country"
}
and
{
_id: ObjectID(), Name: "Full Name", Age: 30 }
```

Create and Insert

You will now look at how databases and collections are created. As explained earlier, the documents in MongoDB are in the JSON format.

First, by issuing the db command you will confirm that the context is the mydbpoc database.

```
> db
mydbpoc
>
```

Now you'll see how to create documents.

The first document complies with the first prototype whereas the second document complies with the second prototype. You have created two documents named *user1* and *user2*.

```
> user1 = {FName: "Test", LName: "User", Age:30, Gender: "M", Country: "US"}
{
        "FName" : "Test",
        "LName" : "User",
        "Age" : 30,
        "Gender" : "M",
        "Country" : "US"
}
> user2 = {Name: "Test User", Age:45, Gender: "F", Country: "US"}
{ "Name" : "Test User", "Age" : 45, "Gender" : "F", "Country" : "US" }
>
```

You will next add both these documents (*user1* and *user2)* to the users collection in the following order of operations:

```
> db.users.insert(user1)
> db.users.insert(user2)
>
```

The above operation will not only insert the two documents to the users collection but it will also create the collection as well as the database. The same can be verified using the show collections and show dbs commands.

As mentioned, `show dbs` will display the list of databases.

```
> show dbs
admin     0.078GB
local     0.078GB
mydb      0.078GB
mydbproc  0.078GB
```

And `show collections` will display the list of collection in the current database.

```
> show collections
system.indexes
users
>
```

Along with the collection `users`, the `system.indexes` collection also gets displayed. This `system.indexes` collection is created by default when the database is created. It manages the information of all the indexes of all collections within the database.

Executing the command `db.users.find()` will display the documents in the `users` collection.

```
> db.users.find()
{ "_id" : ObjectId("5450c048199484c9a4d26b0a"), "FName" : "Test", "LName" : "User",
"Age" : 30, "Gender": "M", "Country" : "US" }
{ "_id" : ObjectId("5450c05d199484c9a4d26b0b"), "Name" : "Test", User", "Age" : 45,
"Gender" : "F", "Country" : "US" }

>
```

You can see that the two documents you created are displayed. In addition to the fields you added to the document, there's an additional `_id` field that is generated for all of the documents.

All documents must have a unique `_id` field. If not explicitly specified by you, the same will be auto-assigned as a unique object ID by MongoDB, as shown in the example above.

You didn't explicitly insert an `_id` field but when you use the `find()` command to display the documents you can see an `_id` field associated with each document.

The reason behind this is by default an index is created on the `_id` field, which can be validated by issuing the `find` command on the `system.indexes` collection.

```
>db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "mydbpoc.users", "name" : "_id_" }
>
```

New indexes can be added or removed from the collection using the `ensureIndex()` and `dropIndex()` commands. We will cover this later in this chapter. By default, an index is created on the `_id` field of all collections. *This default index cannot be dropped.*

Explicitly Creating Collections

In the above example, the first insert operation implicitly created the collection. However, the user can also explicitly create a collection before executing the insert statement.

```
db.createCollection("users")
```

Inserting Documents Using Loop

Documents can also be added to the collection using a *for* loop. The following code inserts users using *for*.

```
> for(var i=1; i<=20; i++) db.users.insert({"Name" : "Test User" + i, "Age": 10+i,
"Gender" : "F", "Country" : "India"})
>
```

In order to verify that the insert is successful, run the `find` command on the collection.

```
> db.users.find()
{ "_id" : ObjectId("52f48cf474f8fdcfcae84f79"), "FName" : "Test", "LName" : "User",
"Age" : 30, "Gender" : "M", "Country" : "US" }
{ "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" : "Test User", "Age" : 45
, "Gender" : "F", "Country" : "US" }
.................
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f8c"), "Name" : "Test User18", "Age" :
28, "Gender" : "F", "Country" : "India" }
Type "it" for more
>
```

Users appear in the collection. Before you go any further, let's understand the "*Type "it" for more*" statement.

The `find` command returns a cursor to the result set. Instead of displaying all documents(which can be thousands or millions of results) in one go on the screen, the cursor displays first 20 documents and waits for the request to iterate (*it*) to display the next 20 and so on until all of the result set is displayed.

The resulting cursor can also be assigned to a variable and then programmatically it can be iterated over using a *while* loop. The cursor object can also be manipulated as an array.

In your case, if you type *"it"* and press Enter, the following will appear:

```
> it
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f8d"), "Name" : "Test User19", "Age" :
29, "Gender" : "F", "Country" : "India" }
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f8e"), "Name" : "Test User20", "Age" :
30, "Gender" : "F", "Country" : "India" }
>
```

Since only two documents were remaining, it displays the remaining two documents.

Inserting by Explicitly Specifying _id

In the previous examples of insert, the `_id` field was not specified, so it was implicitly added. In the following example, you will see how to explicitly specify the `_id` field when inserting the documents within a collection.

While explicitly specifying the `_id` field, you have to keep in mind the uniqueness of the field; otherwise the insert will fail.

The following command explicitly specifies the _id field:

```
> db.users.insert({"_id":10, "Name": "explicit id"})
```

The insert operation creates the following document in the users collection:

```
{ "_id" : 10, "Name" : "explicit id" }
```

This can be confirmed by issuing the following command:

```
>db.users.find()
```

Update

In this section, you will explore the update() command, which is used to update the documents in a collection.

The update() method updates a single document by default. If you need to update all documents that match the selection criteria, you can do so by setting the multi option as true.

Let's begin by updating the values of existing columns. The $set operator will be used for updating the records.

The following command updates the country to UK for all female users:

```
> db.users.update({"Gender":"F"}, {$set:{"Country":"UK"}})
```

To check whether the update has happened, issue a find command to check all the female users.

```
> db.users.find({"Gender":"F"})
{ "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" : "Test User", "Age" : 45
, "Gender" : "F", "Country" : "UK" }
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f7b"), "Name" : "Test User1", "Age" : 11,
"Gender" : "F", "Country" : "India" }
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f7c"), "Name" : "Test User2", "Age" : 12,
"Gender" : "F", "Country" : "India" }
...................
Type "it" for more
>
```

If you check the output, you will see that only the first document record is updated, which is the default behavior of update since no multi option was specified.

Now let's change the update command and include the multi option:

```
>db.users.update({"Gender":"F"},{$set:{"Country":"UK"}},{multi:true})
>
```

Issue the find command again to check whether the country has been updated for all the female employees or not. Issuing the find command will return the following output:

```
> db.users.find({"Gender":"F"})
{ "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" : "Test User", "Age" : 45,
"Gender" : "F", "Country" : "UK" }
..............
Type "it" for more
>
```

As you can see, the country is updated to UK for all records that matched the criteria.

When working in a real-world application, you may come across a schema evolution where you might end up adding or removing fields from the documents. Let's see how to perform these alterations in the MongoDB database.

The `update()` operations can be used at the document level, which helps in updating either a single document or set of documents within a collection.

Next, let's look at how to add new fields to the documents. In order to add fields to the document, use the `update()` command with the `$set` operator and the `multi` option.

If you use a field name with `$set`, which is non-existent, then the field will be added to the documents. The following command will add the field `company` to all the documents:

```
> db.users.update({},{$set:{"Company":"TestComp"}},{multi:true})
>
```

Issuing `find` command against the user's collection, you will find the new field added to all documents.

```
> db.users.find()
{ "Age" : 30, "Company" : "TestComp", "Country" : "US", "FName" : "Test", "Gender" : "M",
"LName" : "User", "_id" : ObjectId("52f48cf474f8fdcfcae84f79") }
{ "Age" : 45, "Company" : "TestComp", "Country" : "UK", "Gender" : "F", "Name" : "Test
User", "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a") }
{ "Age" : 11, "Company" : "TestComp", "Country" : "UK", "Gender" : "F", ...................
Type "it" for more
>
```

If you execute the `update()` command with fields existing in the document, it will update the field's value; however, if the field is not present in the document, then the field will be added to the documents.

You will next see how to use the same `update()` command with the `$unset` operator to remove fields from the documents.

The following command will remove the field `Company` from all the documents:

```
> db.users.update({},{$unset:{"Company":""}},{multi:true})
>
```

This can be checked by issuing the `find()` command against the `Users` collection. You can see that the `Company` field has been deleted from the documents.

```
> db.users.find()
{ "Age" : 30, "Country" : "US", "FName" : "Test", "Gender" : "M", "LName" : "User", "_id" :
ObjectId("52f48cf474f8fdcfcae84f79") }
.............
Type "it" for more
```

Delete

To delete documents in a collection, use the `remove ()` method. If you specify a selection criterion, only the documents meeting the criteria will be deleted. If no criteria is specified, all of the documents will be deleted.

The following command will delete the documents where *Gender = 'M'*:

```
> db.users.remove({"Gender":"M"})
>
```

The same can be verified by issuing the `find()` command on `Users`:

```
> db.users.find({"Gender":"M"})
>
```

No documents are returned.
The following command will delete all documents:

```
> db.users.remove({})
> db.users.find()
>
```

As you can see, no documents are returned.
Finally, if you want to drop the collection, the following command will drop the collection:

```
> db.users.drop()
true
>
```

In order to validate whether the collection is dropped or not, issue the `show collections` command.

```
> show collections
system.indexes
>
```

As you can see, the collection name is not displayed, confirming that the collection has been removed from the database.

Having covered the basic Create, Update, and Delete operations, the next section will show you how to perform Read operations.

Read

In this part of the chapter, you will look at various examples illustrating the querying functionality available as part of MongoDB that enables you to read the stored data from the database.

In order to start with basic querying, first create the `users` collection and insert data following the `insert` command.

```
>  user1 = {FName: "Test", LName: "User", Age:30, Gender: "M", Country: "US"}
{
        "FName" : "Test",
        "LName" : "User",
        "Age" : 30,
        "Gender" : "M",
        "Country" : "US"
}
```

```
> user2 = {Name: "Test User", Age:45, Gender: "F", Country: "US"}
{ "Name" : "Test User", "Age" : 45, "Gender" : "F", "Country" : "US" }
> db.users.insert(user1)
> db.users.insert(user2)
> for(var i=1; i<=20; i++) db.users.insert({"Name" : "Test User" + i, "Age": 10+i,
"Gender" : "F", "Country" : "India"})
```

Now let's start with basic querying. The `find()` command is used to retrieve data from the database. Firing a `find()` command returns all the documents within the collection.

```
> db.users.find()
{ "_id" : ObjectId("52f4a823958073ea07e15070"), "FName" : "Test", "LName" : "User",
"Age" : 30, "Gender" : "M", "Country" : "US" }
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User", "Age" : 45,
"Gender" : "F", "Country" : "US" }
......
{ "_id" : ObjectId("52f4a83f958073ea07e15083"), "Name" : "Test User18", "Age" :28,
"Gender" : "F", "Country" : "India" }
Type "it" for more
>
```

Query Documents

A rich query system is provided by MongoDB. Query documents can be passed as a parameter to the `find()` method to filter documents within a collection.

A query document is specified within open "{" and closed "}" curly braces. A query document is matched against all of the documents in the collection before returning the result set.

Using the `find()` command without any query document or an empty query document such as `find({})` returns all the documents within the collection.

A query document can contain selectors and projectors.

A selector is like a where condition in SQL or a filter that is used to filter out the results.

A projector is like the select condition or the selection list that is used to display the data fields.

Selector

You will now see how to use the selector. The following command will return all the female users:

```
> db.users.find({"Gender":"F"})
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User", "Age" : 45,
"Gender" : "F", "Country" : "US" }
.............
{ "_id" : ObjectId("52f4a83f958073ea07e15084"), "Name" : "Test User19", "Age" :29,
"Gender" : "F", "Country" : "India" }
Type "it" for more
>
```

Let's step it up a notch. MongoDB also supports operators that merge different conditions together in order to refine your search on the basis of your requirements.

Let's refine the above query to now look for female users from India. The following command will return the same:

```
> db.users.find({"Gender":"F", $or: [{"Country":"India"}]})
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1", "Age" : 11,
"Gender" : "F", "Country" : "India" }
...........
{ "_id" : ObjectId("52f4a83f958073ea07e15085"), "Name" : "Test User20", "Age" :30,
"Gender" : "F", "Country" : "India" }
>
```

Next, if you want to find all female users who belong to either India or US, execute the following command:

```
>db.users.find({"Gender":"F",$or:[{"Country":"India"},{"Country":"US"}]})
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User", "Age" : 45,
"Gender" : "F", "Country" : "US" }
.............
{ "_id" : ObjectId("52f4a83f958073ea07e15084"), "Name" : "Test User19", "Age" :29,
"Gender" : "F", "Country" : "India" }
Type "it" for more
```

For aggregation requirements, the aggregate functions need to be used. Next, you'll learn how to use the `count()` function for aggregation.

In the above example, instead of displaying the documents, you want to find out a count of female users who stay in either India or the US. So execute the following command:

```
>db.users.find({"Gender":"F",$or:[{"Country":"India"}, {"Country":"US"}]}).count()
21
>
```

If you want to find a count of users irrespective of any selectors, execute the following command:

```
> db.users.find().count()
22
>
```

Projector

You have seen how to use selectors to filter out documents within the collection. In the above example, the `find()` command returns all fields of the documents matching the selector.

Let's add a projector to the query document where, in addition to the selector, you will also mention specific details or fields that need to be displayed.

Suppose you want to display the first name and age of all female employees. In this case, along with the selector, a projector is also used.

Execute the following command to return the desired result set:

```
> db.users.find({"Gender":"F"}, {"Name":1,"Age":1})
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User", "Age" : 45 }
..........
Type "it" for more
>
```

sort()

In MongoDB, the sort order is specified as follows: 1 for ascending and -1 for descending sort.

If in the above example you want to sort the records by ascending order of *age*, you execute the following command:

```
>db.users.find({"Gender":"F"}, {"Name":1,"Age":1}).sort({"Age":1})
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1", "Age" : 11 }
{ "_id" : ObjectId("52f4a83f958073ea07e15073"), "Name" : "Test User2", "Age" : 12 }
{ "_id" : ObjectId("52f4a83f958073ea07e15074"), "Name" : "Test User3", "Age" : 13 }
..............
{ "_id" : ObjectId("52f4a83f958073ea07e15085"), "Name" : "Test User20", "Age" :30 }
Type "it" for more
```

If you want to display the records in descending order by *name* and ascending order by *age*, you execute the following command:

```
>db.users.find({"Gender":"F"},{"Name":1,"Age":1}).sort({"Name":-1,"Age":1})
{ "_id" : ObjectId("52f4a83f958073ea07e1507a"), "Name" : "Test User9", "Age" : 19 }
............
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1", "Age" : 11 }
Type "it" for more
```

limit()

You will now look at how you can limit the records in your result set. For example, in huge collections with thousands of documents, if you want to return only five matching documents, the `limit` command is used, which enables you to do exactly that.

Returning to your previous query of female users who live in either India or US, say you want to limit the result set and return only two users. The following command needs to be executed:

```
>db.users.find({"Gender":"F",$or:[{"Country":"India"},{"Country":"US"}]}).limit(2)
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User", "Age" : 45,
"Gender" : "F", "Country" : "US" }
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1", "Age" : 11,
"Gender" : "F", "Country" : "India" }
```

skip()

If the requirement is to skip the first two records and return the third and fourth user, the `skip` command is used. The following command needs to be executed:

```
>db.users.find({"Gender":"F",$or:[{"Country":"India"}, {"Country":"US"}]}).limit(2).skip(2)
{ "_id" : ObjectId("52f4a83f958073ea07e15073"), "Name" : "Test User2", "Age" : 12,
"Gender" : "F", "Country" : "India" }
{ "_id" : ObjectId("52f4a83f958073ea07e15074"), "Name" : "Test User3", "Age" : 13,
"Gender" : "F", "Country" : "India" }
>
```

findOne()

Similar to `find()` is the `findOne()` command. The `findOne()` method can take the same parameters as `find()`, but rather then returning a cursor, it returns a single document. Say you want to return one female user who stays in either India or US. This can be achieved using the following command:

```
> db.users.findOne({"Gender":"F"}, {"Name":1,"Age":1})
{
        "_id" : ObjectId("52f4a826958073ea07e15071"),
        "Name" : "Test User",
        "Age" : 45
}
>
```

Similarly, if you want to return the first record irrespective of any selector in that case, you can use `findOne()` and it will return the first document in the collection.

```
> db.users.findOne()
{
        "_id" : ObjectId("52f4a823958073ea07e15070"),
        "FName" : "Test",
        "LName" : "User",
        "Age" : 30,
        "Gender" : "M",
        "Country" : "US"}
```

Using Cursor

When the `find()` method is used, MongoDB returns the results of the query as a cursor object. In order to display the result, the mongo shell iterates over the returned cursor.

MongoDB enables the users to work with the Cursor object of the `find` method. In the next example, you will see how to store the cursor object in a variable and manipulate it using a *while* loop.

Say you want to return all the users in the US. In order to do so, you created a variable, assigned the output of `find()` to the variable, which is a cursor, and then using the *while* loop you iterate and print the output.

The code snippet is as follows:

```
> var c = db.users.find({"Country":"US"})
> while(c.hasNext()) printjson(c.next())
{
        "_id" : ObjectId("52f4a823958073ea07e15070"),
        "FName" : "Test",
        "LName" : "User",
        "Age" : 30,
        "Gender" : "M",
        "Country" : "US"
}
```

```
{
        "_id" : ObjectId("52f4a826958073ea07e15071"),
        "Name" : "Test User",
        "Age" : 45,
        "Gender" : "F",
        "Country" : "US"
}
>
```

The next() function returns the next document. The hasNext() function returns true if a document exists, and printjson() renders the output in JSON format.

The variable to which the cursor object is assigned can also be manipulated as an array. If, instead of looping through the variable, you want to display the document at array index 1, you can run the following command:

```
> var c = db.users.find({"Country":"US"})
> printjson(c[1])
{
        "_id" : ObjectId("52f4a826958073ea07e15071"),
        "Name" : "Test User",
....     "Gender" : "F",
       "Country" : "US"}
>
```

explain()

The explain() function can be used to see what steps the MongoDB database is running while executing a query. Starting from version 3.0, the output format of the function and the parameter that is passed to the function have changed. It takes an optional parameter called verbose, which determines what the explain output should look like. The following are the verbosity modes: allPlansExecution, executionStats, and queryPlanner. The default verbosity mode is queryPlanner, which means if nothing is specified, it defaults to queryPlanner.

The following code covers the steps executed when filtering on the username field:

```
> db.users.find({"Name":"Test User"}).explain("allPlansExecution")

"queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "mydbproc.users",
        "indexFilterSet" : false,
        "parsedQuery" : {
                "$and" : [ ]
        },
        "winningPlan" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "$and" : [ ]
                },
                "direction" : "forward"
        },
        "rejectedPlans" : [ ]
},
```

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 20,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 20,
        "executionStages" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "$and" : [ ]
                },
                "nReturned" : 20,
                "executionTimeMillisEstimate" : 0,
                "works" : 22,
                "advanced" : 20,
                "needTime" : 1,
                "needFetch" : 0,
                "saveState" : 0,
                "restoreState" : 0,
                "isEOF" : 1,
                "invalidates" : 0,
                "direction" : "forward",
                "docsExamined" : 20
        },
        "allPlansExecution" : [ ]
},
"serverInfo" : {
        "host" : " ANOC9",
        "port" : 27017,
        "version" : "3.0.4",
        "gitVersion" : "534b5a3f9d10f00cd27737fbcd951032248b5952"
},
"ok" : 1
```

As you can see, the `explain()` output returns information regarding `queryPlanner`, `executionStats`, and `serverInfo`. As highlighted above, the information the output returns depends on the verbosity mode selected.

You have seen how to perform basic querying, sorting, limiting, etc. You also saw how to manipulate the result set using a *while* loop or as an array. In the next section, you will take a look at indexes and how you can use them in your queries.

Using Indexes

Indexes are used to provide high performance read operations for queries that are used frequently. By default, whenever a collection is created and documents are added to it, an index is created on the `_id` field of the document.

In this section, you will look at how different types of indexes can be created. Let's begin by inserting 1 million documents using *for* loop in a new collection called `testindx`.

```
>for(i=0;i<1000000;i++){db.testindx.insert({"Name":"user"+i,"Age":Math.floor(Math.
random()*120)})}
```

Next, issue the `find()` command to fetch a *Name* with value of *user101*. Run the `explain()` command to check what steps MongoDB is executing in order to return the result set.

```
> db.testindx.find({"Name":"user101"}).explain("allPlansExecution")

{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "mydbproc.testindx",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "Name" : {
                                "$eq" : "user101"
                        }
                },
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "Name" : {
                                        "$eq" : "user101"
                                }
                        },
                        "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 645,
                "totalKeysExamined" : 0,
                "totalDocsExamined" : 1000000,
                "executionStages" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "Name" : {
                                        "$eq" : "user101"
                                }
                        },
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 20,
                        "works" : 1000002,
                        "advanced" : 1,
                        "needTime" : 1000000,
                        "needFetch" : 0,
                        "saveState" : 7812,
                        "restoreState" : 7812,
                        "isEOF" : 1,
                        "invalidates" : 0,
```

```
                "direction" : "forward",
                "docsExamined" : 1000000
        },
        "allPlansExecution" : [ ]
    },
    "serverInfo" : {
        "host" : " ANOC9",
        "port" : 27017,
        "version" : "3.0.4",
        "gitVersion" : "534b5a3f9d10f00cd27737fbcd951032248b5952"
    },
    "ok" : 1
```

As you can see, the database scanned the entire table. This has a significant performance impact and it is happening because there are no indexes.

Single Key Index

Let's create an index on the `Name` field of the document. Use `ensureIndex()` to create the index.

```
> db.testindx.ensureIndex({"Name":1})
```

The index creation will take few minutes depending on the server and the collection size.

Let's run the same query that you ran earlier with `explain()` to check what the steps the database is executing post index creation. Check the `n`, `nscanned`, and `millis` fields in the output.

```
>db.testindx.find({"Name":"user101"}).explain("allPathsExecution")

{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "mydbproc.testindx",
        "indexFilterSet" : false,
        "parsedQuery" : {
            "Name" : {
                "$eq" : "user101"
            }
        },
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "Name" : 1
                },
                "indexName" : "Name_1",
                "isMultiKey" : false,
                "direction" : "forward",
```

```
                                "indexBounds" : {
                                        "Name" : [
                                                "[\"user101\", \"user101\"]"
                                        ]
                                }
                        }
                },
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 0,
                "totalKeysExamined" : 1,
                "totalDocsExamined" : 1,
                "executionStages" : {
                        "stage" : "FETCH",
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 0,
                        "works" : 2,
                        "advanced" : 1,
                        "needTime" : 0,
                        "needFetch" : 0,
                        "saveState" : 0,
                        "restoreState" : 0,
                        "isEOF" : 1,
                        "invalidates" : 0,
                        "docsExamined" : 1,
                        "alreadyHasObj" : 0,
                        "inputStage" : {
                                "stage" : "IXSCAN",
                                "nReturned" : 1,
                                "executionTimeMillisEstimate" : 0,
                                "works" : 2,
                                "advanced" : 1,
                                "needTime" : 0,
                                "needFetch" : 0,
                                "saveState" : 0,
                                "restoreState" : 0,
                                "isEOF" : 1,
                                "invalidates" : 0,
                                "keyPattern" : {
                                        "Name" : 1
                                },
                                "indexName" : "Name_1",
                                "isMultiKey" : false,
                                "direction" : "forward",
                                "indexBounds" : {
                                        "Name" : [
                                                "[\"user101\", \"user101\"]"
                                        ]
                                },
```

```
                                "keysExamined" : 1,
                                "dupsTested" : 0,
                                "dupsDropped" : 0,
                                "seenInvalidated" : 0,
                                "matchTested" : 0
                        }
                },
                "allPlansExecution" : [ ]
        },
        "serverInfo" : {
                "host" : "ANOC9",
                "port" : 27017,
                "version" : "3.0.4",
                "gitVersion" : "534b5a3f9d10f00cd27737fbcd951032248b5952"
        },
        "ok" : 1
}
>
```

As you can see in the results, there is no table scan. The index creation makes a significant difference in the query execution time.

Compound Index

When creating an index, you should keep in mind that the index covers most of your queries. If you sometimes query only the `Name` field and at times you query both the `Name` and the `Age` field, creating a compound index on the `Name` and `Age` fields will be more beneficial than an index that is created on either of the fields because the compound index will cover both queries.

The following command creates a compound index on fields `Name` and `Age` of the collection `testindx`.

```
> db.testindx.ensureIndex({"Name":1, "Age": 1})
```

Compound indexes help MongoDB execute queries with multiple clauses more efficiently. When creating a compound index, it is also very important to keep in mind that the fields that will be used for exact matches (e.g. `Name: "S1"`) come first, followed by fields that are used in ranges (e.g. `Age: {"$gt":20}`).

Hence the above index will be beneficial for the following query:

```
>db.testindx.find({"Name": "user5","Age":{"$gt":25}}).explain("allPlansExecution")

{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "mydbproc.testindx",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "$and" : [
                                {
                                        "Name" : {
                                                "$eq" : "user5"
                                        }
                                },
```

```
                                {
                                        "Age" : {
                                                "$gt" : 25
                                        }
                                }
                        ]
                },
                "winningPlan" : {
                        "stage" : "KEEP_MUTATIONS",
                        "inputStage" : {
                                "stage" : "FETCH",
                                "filter" : {
                                        "Age" : {
                                                "$gt" : 25
                                        }
                                },
                                ........................
                                        "indexBounds" : {
                                                "Name" : [
                                                        "[\"user5\", \"user5\"
                },
                "rejectedPlans" : [
                        {
                                "stage" : "FETCH",
.................................................
                                        "indexName" : "Name_1_Age_1",
                                        "isMultiKey" : false,
                                        "direction" : "forward",
.................................................
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 0,
                "totalKeysExamined" : 1,
                "totalDocsExamined" : 1,
.................................................
                        "inputStage" : {
                                "stage" : "FETCH",
                                "filter" : {
                                        "Age" : {
                                                "$gt" : 25
                                        }
                                },
                                "nReturned" : 1,
                                "executionTimeMillisEstimate" : 0,
                                "works" : 2,
                                "advanced" : 1,
                "allPlansExecution" : [
                        {
                                "nReturned" : 1,
                                "executionTimeMillisEstimate" : 0,
```

```
                                "totalKeysExamined" : 1,
                                "totalDocsExamined" : 1,
                                "executionStages" : {
.....................................................
        "serverInfo" : {
                "host" : " ANOC9",
                "port" : 27017,
                "version" : "3.0.4",
                "gitVersion" : "534b5a3f9d10f00cd27737fbcd951032248b5952"
        },
        "ok" : 1
}
>
```

Support for sort Operations

In MongoDB, a sort operation that uses an indexed field to sort documents provides the greatest performance.

As in other databases, indexes in MongoDB have an order due to this. If an index is used to access documents, it returns results in the same order as the index.

A compound index needs to be created when sorting on multiple fields. In a compound index, the output can be in the sorted order of either an index prefix or the full index.

The index prefix is a subset of the compound index, which contains one or more fields from the start of the index.

For example, the following are the index prefix of the compound index: `{ x:1, y: 1, z: 1}`.

The sort operation can be on any of the combinations of index prefix like `{x: 1}, {x: 1, y: 1}`.

A compound index can only help with sorting if it is a prefix of the sort.

For example, a compound index on `Age`, `Name`, and `Class`, like

```
> db.testindx.ensureIndex({"Age": 1, "Name": 1, "Class": 1})
```

will be useful for the following queries:

```
> db.testindx.find().sort({"Age":1})
> db.testindx.find().sort({"Age":1,"Name":1})
> db.testindx.find().sort({"Age":1,"Name":1, "Class":1})
```

The above index won't be of much help in the following query:

```
> db.testindx.find().sort({"Gender":1, "Age":1, "Name": 1})
```

You can diagnose how MongoDB is processing a query by using the `explain()` command.

Unique Index

Creating index on a field doesn't ensure uniqueness, so if an index is created on the `Name` field, then two or more documents can have the same names. However, if uniqueness is one of the constraints that needs to be enabled, the `unique` property needs to be set to *true* when creating the index.

First, let's drop the existing indexes.

```
>db.testindx.dropIndexes()
```

The following command will create a unique index on the Name field of the testindx collection:

```
> db.testindx.ensureIndex({"Name":1},{"unique":true})
```

Now if you try to insert duplicate names in the collection as shown below, MongoDB returns an error and does not allow insertion of duplicate records:

```
> db.testindx.insert({"Name":"uniquename"})
> db.testindx.insert({"Name":"uniquename"})
"E11000 duplicate key error index: mydbpoc.testindx.$Name_1 dup key: { : "uniquename" }"
```

If you check the collection, you'll see that only the first uniquename was stored.

```
> db.testindx.find({"Name":"uniquename"})
{ "_id" : ObjectId("52f4b3c3958073ea07f092ca"), "Name" : "uniquename" }
>
```

Uniqueness can be enabled for compound indexes also, which means that although individual fields can have duplicate values, the combination will always be unique.

For example, if you have a unique index on {"name":1, "age":1},

```
> db.testindx.ensureIndex({"Name":1, "Age":1},{"unique":true})
>
```

then the following inserts will be permissible:

```
> db.testindx.insert({"Name":"usercit"})
> db.testindx.insert({"Name":"usercit", "Age":30})
```

However, if you execute

```
> db.testindx.insert({"Name":"usercit", "Age":30})
```

it'll throw an error like

```
E11000 duplicate key error index: mydbpoc.testindx.$Name_1_Age_1
dup key: { : "usercit", : 30.0 }
```

You may create the collection and insert the documents first and then create an index on the collection. If you create a unique index on the collection that might have duplicate values in the fields on which the index is being created, the index creation will fail.

To cater to this scenario, MongoDB provides a dropDups option. The dropDups option saves the first document found and remove any subsequent documents with duplicate values.

The following command will create a unique index on the name field and will delete any duplicate documents:

```
>db.testindx.ensureIndex({"Name":1},{"unique":true, "dropDups":true})
>
```

system.indexes

Whenever you create a database, by default a `system.indexes` collection is created. All of the information about a database's indexes is stored in the `system.indexes` collection. This is a reserved collection, so you cannot modify its documents or remove documents from it. You can manipulate it only through `ensureIndex` and the `dropIndexes` database commands.

Whenever an index is created, its meta information can be seen in `system.indexes`. The following command can be used to fetch all the index information about the mentioned collection:

```
db.collectionName.getIndexes()
```

For example, the following command will return all indexes created on the `testindx` collection:

```
> db.testindx.getIndexes()
```

dropIndex

The `dropIndex` command is used to remove the index.

The following command will remove the `Name` field index from the `testindx` collection:

```
> db.testindx.dropIndex({"Name":1})
{ "nIndexesWas" : 3, "ok" : 1 }
>
```

reIndex

When you have performed a number of insertions and deletions on the collection, you may have to rebuild the indexes so that the index can be used optimally. The `reIndex` command is used to rebuild the indexes.

The following command rebuilds all the indexes of a collection. It will first drop the indexes, including the default index on the `_id` field, and then it will rebuild the indexes.

```
db.collectionname.reIndex()
```

The following command rebuilds the indexes of the `testindx` collection:

```
> db.testindx.reIndex()
{
        "nIndexesWas" : 2,
        "msg" : "indexes dropped for collection",
        "nIndexes" : 2,
        ..............
        "ok" : 1
}
>
```

We will be discussing in detail the different types of indexes available in MongoDB in the next chapter.

How Indexing Works

MongoDB stores indexes in a *BTree* structure, so range queries are automatically supported.

If multiple selection criteria are used in a query, MongoDB tries to find the best single index to select a candidate set. After that, it sequentially iterates through the set to evaluate the other criteria.

When the query is executed for the first time, MongoDB creates multiple execution plans for each index that is available for the query. It lets the plans execute within certain number of ticks in turns, until the plan that executes the fastest finishes. The result is then returned to the system, which remembers the index that was used by the fastest execution plan.

For subsequent queries, the remembered index will be used until some certain number of updates has happened within the collection. After the updating limit is crossed, the system will again follow the process to find out the best index that is applicable at that time.

The reevaluation of the query plans will happen when either of the following events has occurred:

- The collection receives 1,000 write operations.
- An index is added or dropped.
- A restart of the mongod process happens.
- A reindexing for rebuilding the index happens.

If you want to override MongoDB's default index selection, the same can be done using the `hint()` method.

The index filter is introduced in version 2.6. It is made of indexes that an optimizer will evaluate for a query, including the query, projections, and the sorting. MongoDB will use the index as provided by the index filter and will ignore the `hint()`.

Before version 2.6, at any point in time MongoDB uses only one index, so you need to ensure that composite indexes exist to match your queries better. This can be done by checking the sort and search criteria of the queries.

Index intersection is introduced in version 2.6. It enables intersection of indexes for fulfilling queries with compound conditions where part of condition is fulfilled by one index and the other part is fulfilled by the other index.

In general, an index intersection is made up of two indexes; however, multiple index intersections can be used for resolving a query. This capability provides better optimization.

As in other databases, index maintenance has a cost attached. Every operation that changes the collection (such as creation, updating, or deletion) has an overhead because the indexes also need to be updated. To maintain an optimal balance, you need to periodically check the effectiveness of having an index that can be measured by the ratio of reads and writes you are doing on the system. Identify the less-used indexes and delete them.

Stepping Beyond the Basics

This section will cover advanced querying using conditional operators and regular expressions in the selector part. Each of these operators and regular expressions provides you with more control over the queries you write and consequently over the information you can fetch from the MongoDB database.

Using Conditional Operators

Conditional operators enable you to have more control over the data you are trying to extract from the database. In this section, you will be focusing on the following operators: `$lt`, `$lte`, `$gt`, `$gte`, `$in`, `$nin`, and `$not`.

The following example assumes a collection named `Students` that contains the following types of documents:

```
{
_id: ObjectID(),
Name: "Full Name",
Age: 30,
Gender: "M",
Class: "C1",
Score: 95
}
```

You will first create the collection and insert few sample documents.

```
>db.students.insert({Name:"S1",Age:25,Gender:"M",Class:"C1",Score:95})
>db.students.insert({Name:"S2",Age:18,Gender:"M",Class:"C1",Score:85})
>db.students.insert({Name:"S3",Age:18,Gender:"F",Class:"C1",Score:85})
>db.students.insert({Name:"S4",Age:18,Gender:"F",Class:"C1",Score:75})
>db.students.insert({Name:"S5",Age:18,Gender:"F",Class:"C2",Score:75})
>db.students.insert({Name:"S6",Age:21,Gender:"M",Class:"C2",Score:100})
>db.students.insert({Name:"S7",Age:21,Gender:"M",Class:"C2",Score:100})
>db.students.insert({Name:"S8",Age:25,Gender:"F",Class:"C2",Score:100})
>db.students.insert({Name:"S9",Age:25,Gender:"F",Class:"C2",Score:90})
>db.students.insert({Name:"S10",Age:28,Gender:"F",Class:"C3",Score:90})
> db.students.find()
{ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25, "Gender" : "M",
"Class" : "C1", "Score" : 95 }
.....................
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" : 28, "Gender" : "F",
"Class" : "C3", "Score" : 90 }
>
```

$lt and $lte

Let's start with the `$lt` and `$lte` operators. They stand for "*less than*" and "*less than or equal to,*" respectively.

If you want to find *all students who are younger than 25* (Age < 25), you can execute the following `find` with a selector:

```
> db.students.find({"Age":{"$lt":25}})
{ "_id" : ObjectId("52f8750ca13cd6a65998734e"), "Name" : "S2", "Age" : 18, "Gender" : "M",
"Class" : "C1", "Score" : 85 }
.........................
{ "_id" : ObjectId("52f87556a13cd6a659987353"), "Name" : "S7", "Age" : 21, "Gender" : "M",
"Class" : "C2", "Score" : 100 }
>
```

If you want to find out *all students who are older than 25* (Age <= 25), execute the following:

```
> db.students.find({"Age":{"$lte":25}})
{ "_id" : ObjectId("52f874faa13cd6a659987334d"), "Name" : "S1", "Age" : 25, "Gender" : "M",
"Class" : "C1", "Score" : 95 }
.....................
{ "_id" : ObjectId("52f87578a13cd6a659987355"), "Name" : "S9", "Age" : 25, "Gender" : "F",
"Class" : "C2", "Score" : 90 }
>
```

$gt and $gte

The `$gt` and `$gte` operators stand for "*greater than*" and "*greater than or equal to,*" respectively.

Let's find out *all of the students with Age > 25*. This can be achieved by executing the following command:

```
> db.students.find({"Age":{"$gt":25}})
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" : 28, "Gender" : "F",
"Class" : "C3", "Score" : 90 }
>
```

If you change the above example to return *students with Age >= 25*, then the command is

```
> db.students.find({"Age":{"$gte":25}})
{ "_id" : ObjectId("52f874faa13cd6a659987334d"), "Name" : "S1", "Age" : 25, "Gender" : "M",
"Class" : "C1", "Score" : 95 }
.....................................
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" : 28, "Gender" : "F",
"Class" : "C3", "Score" : 90 }
>
```

$in and $nin

Let's find *all students who belong to either class C1 or C2*. The command for the same is

```
> db.students.find({"Class":{"$in":["C1","C2"]}})
{ "_id" : ObjectId("52f874faa13cd6a659987334d"), "Name" : "S1", "Age" : 25, "Gender" : "M",
"Class" : "C1", "Score" : 95 }
.............................
{ "_id" : ObjectId("52f87578a13cd6a659987355"), "Name" : "S9", "Age" : 25, "Gender" : "F",
"Class" : "C2", "Score" : 90 }
>
```

The inverse of this can be returned by using `$nin`.

Let's next find *students who don't belong to class C1 or C2*. The command is

```
> db.students.find({"Class":{"$nin":["C1","C2"]}})
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" : 28, "Gender" : "F",
"Class" : "C3", "Score" : 90 }
>
```

Let's next see how you can combine all of the above operators and write a query. Say you want to *find out all students whose gender is either "M" or they belong to class "C1" or 'C2" and whose age is greater than or equal to 25*. This can be achieved by executing the following command:

```
>db.students.find({$or:[{"Gender":"M","Class":{"$in":["C1","C2"]}}], "Age":{"$gte":25}})
{ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25, "Gender" : "M",
"Class" : "C1", "Score" : 95 }
>
```

Regular Expressions

In this section, you will look at how to use regular expressions. Regular expressions are useful in scenarios where you want to *find students with name starting with "A"*.

In order to understand this, let's add three or four more students with different names.

```
> db.students.insert({Name:"Student1", Age:30, Gender:"M", Class: "Biology", Score:90})
> db.students.insert({Name:"Student2", Age:30, Gender:"M", Class: "Chemistry", Score:90})
> db.students.insert({Name:"Test1", Age:30, Gender:"M", Class: "Chemistry", Score:90})
> db.students.insert({Name:"Test2", Age:30, Gender:"M", Class: "Chemistry", Score:90})
> db.students.insert({Name:"Test3", Age:30, Gender:"M", Class: "Chemistry", Score:90})
>
```

Say you want to *find all students with names starting with "St" or "Te" and whose class begins with "Che".* The same can be filtered using regular expressions, like so:

```
> db.students.find({"Name":/(St|Te)*/i, "Class":/(Che)/i})
{ "_id" : ObjectId("52f89ecae451bb7a56e59086"), "Name" : "Student2", "Age" : 30,
"Gender" : "M", "Class" : "Chemistry", "Score" : 90 }
.......................
{ "_id" : ObjectId("52f89f06e451bb7a56e59089"), "Name" : "Test3", "Age" : 30,
"Gender" : "M", "Class" : "Chemistry", "Score" : 90 }
>
```

In order to understand how the regular expression works, let's take the query `"Name":/(St|Te)*/i`.

> **/i** indicates that the regex is case insensitive.
>
> **(St|Te)*** means the Name string must start with either "St" or "Te".
>
> The * at the end means it will match anything after that.

When you put everything together, you are doing a case insensitive match of names that have either "St" or "Te" at the beginning of them. In the regex for the *Class* also the same Regex is issued.

Next, let's complicate the query a bit. Let's combine it with the operators covered above.

Fetch Students with names as student1, student2 and who are male students with age >=25. The command for this is as follows:

```
>db.students.find({"Name":/(student*)/i,"Age":{"$gte":25},"Gender":"M"})
{ "_id" : ObjectId("52f89eb1e451bb7a56e59085"), "Name" : "Student1", "Age" : 30,
 "Gender" : "M", "Class" : "Biology", "Score" : 90 }
{ "_id" : ObjectId("52f89ecae451bb7a56e59086"), "Name" : "Student2", "Age" : 30,
 "Gender" : "M", "Class" : "Chemistry", "Score" : 90 }
```

MapReduce

The MapReduce framework enables division of the task, which in this case is data aggregation across a cluster of computers in order to reduce the time it takes to aggregate the data set. It consists of two parts: Map and Reduce.

Here's a more specific description: MapReduce is a framework that is used to process problems that are highly distributable across enormous datasets and are run using multiple nodes. If all the nodes have the same hardware, these nodes are collectively referred as a cluster; otherwise, it's referred as a grid. This processing can occur on structured data (data stored in a database) and unstructured data (data stored in a file system).

- "Map": In this step, the node that is acting as the master takes the input parameter and divides the big problem into multiple small sub-problems. These sub-problems are then distributed across the worker nodes. The worker nodes might further divide the problem into sub-problems. This leads to a multi-level tree structure. The worker nodes will then work on the sub-problems within them and return the answer back to the master node.
- "Reduce": In this step, all the sub-problems' answers are available with the master node, which then combines all the answers and produce the final output, which is the answer to the big problem you were trying to solve.

In order to understand how it works, let's consider a small example where you will *find out the number of male and female students* in your collection.

This involves the following steps: first you create the `map` and `reduce` functions and then you call the `mapReduce` function and pass the necessary arguments.

Let's start by defining the `map` function:

```
> var map = function(){emit(this.Gender,1);};
>
```

This step takes as input the document and based on the `Gender` field it emits documents of the type `{"F", 1}` or `{"M", 1}`.

Next, you create the `reduce` function:

```
> var reduce = function(key, value){return Array.sum(value);};
>
```

This will group the documents emitted by the `map` function on the key field, which in your example is `Gender`, and will return the sum of values, which in the above example is emitted as "1". The output of the reduce function defined above is a *gender-wise count*.

Finally, you put them together using the `mapReduce` function, like so:

```
> db.students.mapReduce(map, reduce, {out: "mapreducecount1"})
{
        "result" : "mapreducecount1",
        "timeMillis" : 29,
        "counts" : {
                "input" : 15,
                "emit" : 15,
                "reduce" : 2,
                "output" : 2
        },
        "ok" : 1,
}
>
```

This actually is applying the map, reduce function, which you defined on the students collection. The final result is stored in a new collection called mapreducecount1.

In order to vet it, run the find() command on the mapreducecount1 collection, as shown:

```
> db.mapreducecount1.find()
{ "_id" : "F", "value" : 6 }
{ "_id" : "M", "value" : 9 }
>
```

Here's one more example to explain the workings of MapReduce. Let's use MapReduce to *find out a class-wise average score*. As you saw in the above example, you need to first create the map function and then the reduce function and finally you combine them to store the output in a collection in your database. The code snippet is

```
> var map_1 = function(){emit(this.Class,this.Score);};
> var reduce_1 = function(key, value){return Array.avg(value);};
>db.students.mapReduce(map_1,reduce_1, {out:"MR_ClassAvg_1"})
{
        "result" : "MR_ClassAvg_1",
        "timeMillis" : 4,
        "counts" : {
                "input" : 15,  "emit" : 15,
                "reduce" : 3 ,  "output" : 5
        },
        "ok" : 1,
}

> db.MR_ClassAvg_1.find()
{ "_id" : "Biology", "value" : 90 }
{ "_id" : "C1", "value" : 85 }
{ "_id" : "C2", "value" : 93 }
{ "_id" : "C3", "value" : 90 }
{ "_id" : "Chemistry", "value" : 90 }
>
```

The first step is to define the map function, which loops through the collection documents and returns output as {"Class": Score}, for example {"C1":95}. The second step does a grouping on the class and computes the average of the scores for that class. The third step combines the results; it defines the collection to which the map, reduce function needs to be applied and finally it defines where to store the output, which in this case is a new collection called MR_ClassAvg_1.

In the last step, you use find in order to check the resulting output.

aggregate()

The previous section introduced the MapReduce function. In this section, you will get a glimpse of the aggregation framework of MongoDB.

The aggregation framework enables you find out the aggregate value without using the MapReduce function. Performance-wise, the aggregation framework is faster than the MapReduce function. You always need to keep in mind that MapReduce is meant for batch approach and not for real-time analysis.

You will next depict the above two discussed outputs using the aggregate function. First, the output was to *find the count of male and female students*. This can be achieved by executing the following command:

```
> db.students.aggregate({$group:{_id:"$Gender", totalStudent: {$sum: 1}}})
{ "_id" : "F", "totalStudent" : 6 }
{ "_id" : "M", "totalStudent" : 9 }
>
```

Similarly, in order to *find out the class-wise average score*, the following command can be executed:

```
> db.students.aggregate({$group:{_id:"$Class", AvgScore: {$avg: "$Score"}}})
{ "_id" : "Biology", "AvgScore" : 90 }
{ "_id" : "C3", "AvgScore" : 90 }
{ "_id" : "Chemistry", "AvgScore" : 90 }
{ "_id" : "C2", "AvgScore" : 93 }
{ "_id" : "C1", "AvgScore" : 85 }
>
```

Designing an Application's Data Model

In this section, you will look at how to design the data model for an application. The MongoDB database provides two options for designing a data model: the user can either embed related objects within one another, or it can reference each other using ID. In this section, you will explore these options.

In order to understand these options, you will design a blogging application and demonstrate the usage of the two options.

A typical blog application consists of the following scenarios:

You have people posting blogs on different subjects. In addition to the subject categorization, different tags can also be used. As an example, if the category is politics and the post talks about a politician, then that politician's name can be added as a tag to the post. This helps users find posts related to their interests quickly and also lets them link related posts together.

The people viewing the blog can comment on the blog posts.

Relational Data Modeling and Normalization

Before jumping into MongoDB's approach, let's take a little detour into how you would model this in a relational database such as SQL.

In relational databases, the data modelling typically progresses by defining the tables and gradually removing data redundancy to achieve a normal form.

What Is a Normal Form?

In relational databases, a normal form typically begins by creating tables as per the application requirement and then gradually removing redundancy to achieve the highest normal form, which is also termed the third normal form or 3NF.In order to understand this better, let's put the blogging application data in tabular form. The initial data is shown in Figure 6-2.

Author	Posts	Category	Tag	Comments	Commenter

Figure 6-2. *Blogging application initial data*

This data is actually in the first normal form. You will have lots of redundancy because you can have multiple comments against the posts and multiple tags can be associated with the post. The problem with redundancy, of course, is that it introduces the possibility of inconsistency, where various copies of the same data may have different values. To remove this redundancy, you need to further normalize the data by splitting it into multiple tables. As part of this step, you must identify a *key* column that uniquely identifies each row in the table so that you can create links between the tables. The above scenarios when modeled using the 3NF normal forms will look like the RDBMs diagram shown in Figure 6-3.

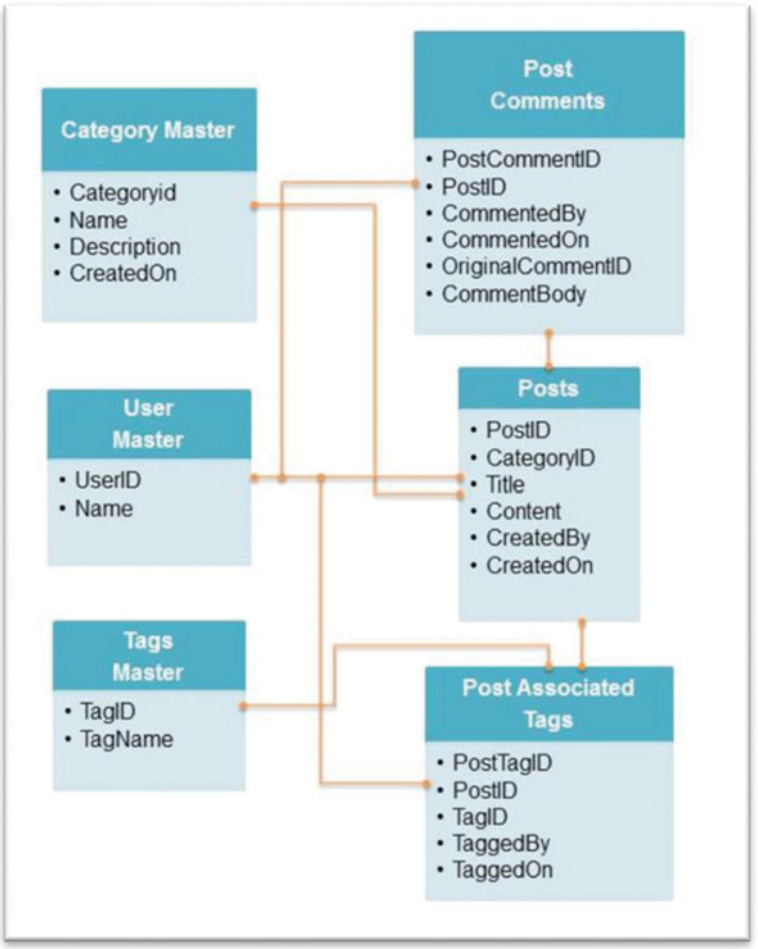

Figure 6-3. *RDBMS diagram*

In this case, you have a data model that is free of redundancy, allowing you to update it without having to worry about updating multiple rows. In particular, you no longer need to worry about *inconsistency* in the data model.

The Problem with Normal Forms

As mentioned, the nice thing about normalization is that it allows for easy updating without any redundancy (i.e. it helps keep the data consistent). Updating a user name means updating the name in the `Users` table.

However, a problem arises when you try to get the data back *out*. For instance, to find all tags and comments associated with posts by a specific user, the relational database programmer uses a JOIN. By using a JOIN, the database returns all data as per the application screen design, but the real problem is what operation the database performs to get that result set.

Generally, any RDBMS reads from a disk and does a seek, which takes well over 99% of the time spent reading a row. When it comes to disk access, random seeks are the enemy. The reason why this is so important in this context is because JOINs typically require random seeks. The JOIN operation is one of the most expensive operations within a relational database. Additionally, if you end up needing to scale your database to multiple servers, you introduce the problem of generating a *distributed join*, a complex and generally slow operation.

MongoDB Document Data Model Approach

As you know, in MongoDB, data is stored in *documents*. Fortunately for us as application designers, this opens up some new possibilities in schema design. Unfortunately for us, it also complicates our schema design process. Now when faced with a schema design problem there's no longer a fixed path of normalized database design, as there is with relational databases. In MongoDB, the schema design depends on the problem you are trying to solve.

If you have to model the above using the MongoDB document model, you might store the blog data in a document as follows:

```
{
                "_id" : ObjectId("509d27069cc1ae293b36928d"),
                "title" : "Sample title",
                "body" : "Sample text.",
                "tags" : [
                        "Tag1",
                        "Tag2",
                        "Tag3",
                        "Tag4"
                ],
                "created_date" : ISODate("2015-07-06T12:41:39.110Z"),
                "author" : "Author 1",
                "category_id" : ObjectId("509d29709cc1ae293b369295"),
                "comments" : [
                        {
                                "subject" : "Sample comment",
                                "body" : "Comment Body",
                                "author " : "author 2",
                                "created_date":ISODate("2015-07-06T13:34:23.929Z")
                        }
                ]}
```

As you can see, you have embedded the comments and tags within a single document only. Alternatively, you could "normalize" the model a bit by referencing the comments and tags by the _id field:

```
// Authors document:
{
"_id": ObjectId("509d280e9cc1ae293b36928e "),
"name": "Author 1",}
// Tags document:
{
"_id": ObjectId("509d35349cc1ae293b369299"),
"TagName": "Tag1",.....}
// Comments document:
{
"_id": ObjectId("509d359a9cc1ae293b3692a0"),
"Author": ObjectId("508d27069cc1ae293b36928d"),
.......
"created_date" : ISODate("2015-07-06T13:34:59.336Z")
}
//Category Document
{
"_id": ObjectId("509d29709cc1ae293b369295"),
"Category": "Catgeory1"......
}
//Posts Document
        {
                "_id" : ObjectId("509d27069cc1ae293b36928d"),
                "title" : "Sample title","body" : "Sample text.",
                "tags" : [      ObjectId("509d35349cc1ae293b369299"),
                        ObjectId("509d35349cc1ae293b36929c")
                ],
                "created_date" : ISODate("2015-07-06T13:41:39.110Z"),
                "author_id" : ObjectId("509d280e9cc1ae293b36928e"),
                "category_id" : ObjectId("509d29709cc1ae293b369295"),
                "comments" : [
                        ObjectId("509d359a9cc1ae293b3692a0"),
                                ]}
```

The remainder of this chapter is devoted to identifying which solution will work in your context (i.e. whether to use referencing or whether to embed).

Embedding

In this section, you will see if embedding will have a positive impact on the performance. Embedding can be useful when you want to fetch some set of data and display it on the screen, such as a page that displays comments associated with the blog; in this case the comments can be embedded in the *Blogs* document.

The benefit of this approach is that since MongoDB stores the documents contiguously on disk, all the related data can be fetched in a single seek.

Apart from this, since JOINs are not supported and you used referencing in this case, the application might do something like the following to fetch the comments data associated with the blog.

1. Fetch the associated `comments _id` from the *blogs* document.
2. Fetch the *comments* document based on the `comments_id` found in the first step.

If you take this approach, which is referencing, not only does the database have to do multiple seeks to find your data, but additional latency is introduced into the lookup since it now takes *two* round trips to the database to retrieve your data.

If the application frequently accesses the comments data along with the blogs, then almost certainly embedding the comments within the *blog* documents will have a positive impact on the performance.

Another concern that weighs in favor of embedding is the desire for *atomicity* and *isolation* in writing data. MongoDB is designed without multi-documents transactions. In MongoDB, the atomicity of the operation is provided only at a single document level so data that needs to be updated together atomically needs to be placed together in a single document.

When you update data in your database, you must ensure that your update either succeeds or fails entirely, never having a "partial success," and that no other database reader ever sees an incomplete write operation.

Referencing

You have seen that embedding is the approach that will provide the best performance in many cases; it also provides data consistency guarantees. However, in some cases, a more normalized model works better in MongoDB.

One reason for having multiple collections and adding references is the increased flexibility it gives when querying the data. Let's understand this with the blogging example mentioned above.

You saw how to use embedded schema, which will work very well when displaying all the data together on a single page (i.e. the page that displays the blog post followed by all of the associated comments).

Now suppose you have a requirement to search for the comments posted by a particular user. The query (using this embedded schema) would be as follows:

```
db.posts.find({'comments.author': 'author2'},{'comments': 1})
```

The result of this query, then, would be documents of the following form:

```
{
                "_id" : ObjectId("509d27069cc1ae293b36928d"),
                "comments" : [        {
                                "subject" : "Sample Comment 1 ",
                                "body" : "Comment1 Body.",
                                "author_id" : "author2",
                                "created_date" : ISODate("2015-07-06T13:34:23.929Z")}...]
        }
                "_id" : ObjectId("509d27069cc1ae293b36928d"),
                "comments" : [
                        {
                                "subject" : "Sample Comment 2",
                                "body" : "Comments Body.",
                                "author_id" : "author2",
                                "created_date" : ISODate("2015-07-06T13:34:23.929Z")
                        }...]}
```

The major drawback to this approach is that you get back *much* more data than you actually need. In particular, you can't ask for just author2's comments; you have to ask for posts that author2 has commented on, which includes all of the other comments on those posts as well. This data will require further filtering within the application code.

On the other hand, suppose you decide to use a normalized schema. In this case you will have three documents: "Authors," "Posts," and "Comments."

The "Authors" document will have Author-specific content such as Name, Age, Gender, etc., and the "Posts" document will have posts-specific details such as post creation time, author of the post, actual content, and the subject of the post.

The "Comments" document will have the post's comments such as CommentedOn date time, created by author, and the text of the comment. This is depicted as follows:

```
// Authors document:
{
"_id": ObjectId("508d280e9cc1ae293b36928e "),
"name": "Jenny",
..........
}
//Posts Document
        {
                "_id" : ObjectId("508d27069cc1ae293b36928d"),.....................
        }
// Comments document:
{
"_id": ObjectId("508d359a9cc1ae293b3692a0"),
"Author": ObjectId("508d27069cc1ae293b36928d"),
 "created_date" : ISODate("2015-07-06T13:34:59.336Z"),
"Post_id": ObjectId("508d27069cc1ae293b36928d"),
..........
}
```

In this scenario, the query to find the comments by "author2" can be fulfilled by a simple `find()` *on the comments collection*:

```
db.comments.find({"author": "author2"})
```

In general, if your application's query pattern is well known, and data tends to be accessed in only one way, an embedded approach works well. Alternatively, if your application may query data in many different ways, or you are not able to anticipate the patterns in which data may be queried, a more "normalized" approach may be better.

For instance, in the above schema, you will be able to sort the comments or return a more restricted set of comments using the *limit, skip* operators. In the embedded case, you're stuck retrieving all the comments in the same order in which they are stored in the post.

Another factor that may weigh in favor of using document references is when you have one-to-many relationships.

For instance, a popular blog with a large amount of reader engagement may have hundreds or even thousands of comments for a given post. In this case, embedding carries significant penalties with it:

- **Effect on read performance**: As the document size increases, it will occupy more memory. The problem with memory is that a MongoDB database caches frequently accessed documents in memory, and the larger the documents become, the lesser the probability of them fitting into memory. This will lead to more page faults while retrieving the documents, which will lead to random disk I/O, which will further slow down the performance.
- **Effect on update performance**: As the size increases and an update operation is performed on such documents to append data, eventually MongoDB is going to need to move the document to an area with more space available. This movement, when it happens, *significantly* slows update performance.

Apart from this, MongoDB documents have a hard size limit of 16MB. Although this is something to be aware of, you will usually run into problems due to memory pressure and document copying well before you reach the 16MB size limit.

One final factor that weighs in favor of using document references is the case of many-to-many or M:N relationships.

For instance, in the above example, there are tags. Each blog can have multiple tags and each tag can be associated to multiple blog entries.

One approach to implement the blogs-tags M:N relationship is to have the following three collections:

- The `Tags` collection, which will store the tags details
- The `Blogs` collection, which will have blogs details
- A third collection, called `Tag-To-Blog Mapping`, which will map between the tags and the blogs

This approach is similar to the one in relational databases, but this will negatively impact the application's performance because the queries will end up doing a lot of application-level "joins."

Alternatively, you can use the embedding model where you embed the tags within the blogs document, but this will lead to data duplication. Although this will simplify the read operation a bit, it will increase the complexity of the update operation, because while updating a tag detail, the user needs to ensure that the updated tag is updated at each and every place where it has been embedded in other blog documents.

Hence for many-to-many joins, a compromise approach is often best, embedding a list of `_id` values rather than the full document:

```
// Tags document:
{
"_id": ObjectId("508d35349cc1ae293b369299"),
"TagName": "Tag1",
..........
}
// Posts document with Tag IDs added as References
//Posts Document
        {                "_id" : ObjectId("508d27069cc1ae293b36928d"),
                "tags" : [
                        ObjectId("509d35349cc1ae293b369299"),
                        ObjectId("509d35349cc1ae293b36929a"),
                        ObjectId("509d35349cc1ae293b36929b"),
                        ObjectId("509d35349cc1ae293b36929c")
                ],.................................
        }
```

Although querying will be a bit complicated, you no longer need to worry about updating a tag everywhere.

In summary, schema design in MongoDB is one of the very early decisions that you need to make, and it is dependent on the application requirements and queries.

As you have seen, when you need to access the data together or you need to make atomic updates, embedding will have a positive impact. However, if you need more flexibility while querying or if you have a many-to-many relationships, using references is a good choice.

Ultimately, the decision depends on the access patterns of your application, and there are no hard-and-fast rules in MongoDB. In the next section, you will learn about various data modelling considerations.

Decisions of Data Modelling

This involves deciding how to structure the documents so that the data is modeled effectively. An important point to decide is whether you need to embed the data or use references to the data (i.e. whether to use embedding or referencing).

This point is best demonstrated with an example. Suppose you have a book review site which has authors and books as well as reviews with threaded comments.

Now the question is how to structure the collections. The decision depends on the number of comments expected on per book and how frequently the read vs. write operations will be performed.

Operational Considerations

In addition to the way the elements interact with each other (i.e. whether to store the documents in an embedded manner or use references), a number of other operational factors are important when designing a data model for the application. These factors are covered in the following sections.

Data Lifecycle Management

This feature needs to be used if your application has datasets that need to be persisted in the database only for a limited time period.

Say you need to retain the data related to the review and comments for a month. This feature can be taken into consideration.

This is implemented by using the Time to Live (TTL) feature of the collection. The TTL feature of the collection ensures that the documents are expired after a period of time.

Additionally, if the application requirement is to work with only the recently inserted documents, using capped collections will help optimize the performance.

Indexes

Indexes can be created to support commonly used queries to increase the performance. By default, an index is created by MongoDB on the `_id` field.

The following are a few points to consider when creating indexes:

- At least 8KB of data space is required by each index.
- For write operations, an index addition has some negative performance impact. Hence for collections with heavy writes, indexes might be expensive because for each insert, the keys must be added to all the indexes.
- Indexes are beneficial for collections with heavy read operations such as where the proportion of read-to-write operations is high. The un-indexed read operations are not affected by an index.

Sharding

One of the important factors when designing the application model is whether to partition the data or not. This is implemented using sharding in MongoDB.

Sharding is also referred as partitioning of data. In MongoDB, a collection is partitioned with its documents distributed across cluster of machines, which are referred as shards. This can have a significant impact on the performance. We will discuss sharding more in Chapter tk.

A Large Number of Collections

The design considerations for having multiple collections vs. storing data in a single collection are the following:

- There is no performance penalty in choosing multiple collections for storing data.
- Having distinct collections for different types of data can have performance improvements in high-throughput batch processing applications.

When you are designing models that have a large number of collections, you need to take into consideration the following behaviors:

- A certain minimum overhead of few kilobytes is associated with each collection.
- At least 8KB of data space is required by each index, including the `_id` index.

You know by now that the metadata for each database is stored in the `<database>.ns` file. Each collection and index has its own entry in the namespace file, so you need to consider the limits_on_the_size_of_namespace files when deciding to implement a large number of collections.

Growth of the Document

Few updates, such as pushing an element to an array, adding new fields, etc., can lead to an increase in the document size, which can lead to the movement of the document from one slot to another in order to fit in the document. This process of document relocation is both resource and time consuming. Although MongoDB provides padding to minimize the relocation occurrences, you may need to handle the document growth manually.

Summary

In this chapter you learned the basic CRUD operations plus advanced querying capabilities. You also examined the two ways of storing and retrieving data: embedding and referencing.

In the following chapter, you will learn about the MongoDB architecture, its core components, and features.